sp-6. リストと繰り返し処理

（Scheme プログラミング）

URL: https://www.kkaneko.jp/pro/scheme/index.html

金子邦彦

アウトライン

6-1 リストと繰り返し処理

6-2 パソコン演習

6-3 課題

本日の内容

1. リストを扱う関数の書き方について
   - 再帰
   - cond 文との組み合わせ
   - リストの要素に対する繰り返し処理
2. 再帰を使ったプログラムに慣れ，自力で読み書きできるようになる

リストと繰り返し処理

リストを扱うプログラムでは

リストの要素の数だけ， 同じ処理を繰り返す

ことが多い

- 長さが１０のリストなら，処理を１０回繰り返したい

⇒　「**再帰**」のテクニック（次ページ）

再帰

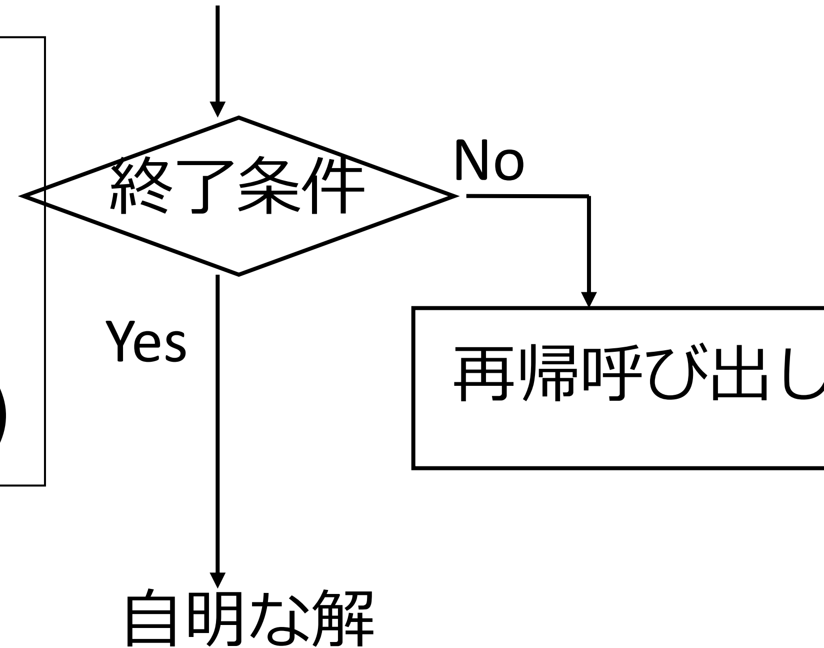
再帰関数のパターン

```
(define (foo パラメータの並び)
  (cond
    [終了条件　自明の答]
    [else (foo 新たなパラメータの並び)]))
```

- 関数(上では foo)の内部に， 同じ関数 foo が登場
  - foo の実行が繰り返される

再帰での終了条件

```
(define (foo ...)
  (cond
    [終了条件　自明の答]
    [else (foo 新たなパラメータ)]))
```



- 条件文（cond）と組み合わせ
  - 繰り返しのたびに「終了条件」の真偽が判定される
  - 「終了条件」が満足されるまで，処理が繰り返される

リストでの繰り返しと終了条件

- ある終了条件（例えば、リストが empty になるなど）が満足されたら，処理を終える
- リストの rest をとりながら，処理を繰り返すことが多い

パソコン演習

Database Lab.

パソコン演習の進め方

Database Lab.

- 資料を見ながら，「例題」を行ってみる
- 各自，「課題」に挑戦する
- 自分のペースで先に進んで構いません

DrScheme の使用

- DrScheme の起動

  プログラム　→ PLT Scheme → DrScheme

- 今日の演習では「Intermediate Student」に設定

  Language
  → Choose Language
  → Intermediate Student
  → Execute ボタン

例題１．リストの総和

- 数のリスト a-list から総和を求める関数 list-sum を作り，実行する
  - $x_1, x_2, \ldots, x_n$ のリストに対して，総和 $x_1 + x_2 + \ldots + x_n$ を求める

「例題１．リストの総和」の手順

1. 次を「定義用ウインドウ」で，実行しなさい
   - 入力した後に，Execute ボタンを押す

```
(define (list-sum a-list)
  (cond
    [(empty? a-list) 0]
    [else (+ (first a-list)
             (list-sum (rest a-list)))]))
```

2. その後，次を「実行用ウインドウ」で実行しなさい

```
(sum (list 1 2 3))
(sum (list 11 12 13))
```

☆　次は，例題２に進んでください

Untitled - DrScheme

File Edit Windows Show Language Scheme Help

Untitled
(define ...)
Save
Check Syntax Step Execute Break

```
(define (list-sum a-list)
  (cond
    [(empty? a-list) 0]
    [else (+ (first a-list)
             (list-sum (rest a-list)))]))
```

```
>
```

まず，Scheme のプログラムを
コンピュータに読み込ませている

3:3 Unlocked not running

Untitled - DrScheme

File Edit Windows Show Language Scheme Help

Untitled (define ...) Save Check Syntax Step Execute Break

```
(define (list-
  (cond
    [(empty? a
    [else (+
```

これは，

(list-sum (list 1 2 3))

と書いて，a-list の値を

(list 1 2 3) に設定しての実行

```
> (list-sum (list 1 2 3))
6
> (list-s
36
>
```

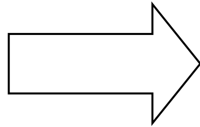
実行結果である「6」が

表示される

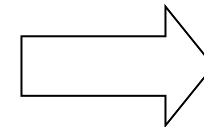
7:3 Unlocked not running

入力と出力

(list 1 2 3)



6



入力は
１つのリスト

出力は
１つの数値

list-sum 関数

「関数である」ことを
示すキーワード

関数の名前

```
(define (list-sum a-list)
  (cond
    [(empty? a-list) 0]
    [else (+ (first a-list)
             (list-sum (rest a-list)))]))
```

値を1つ受け取る（入力）

a-list の値から
リストの総和を求める（出力）

リストの総和

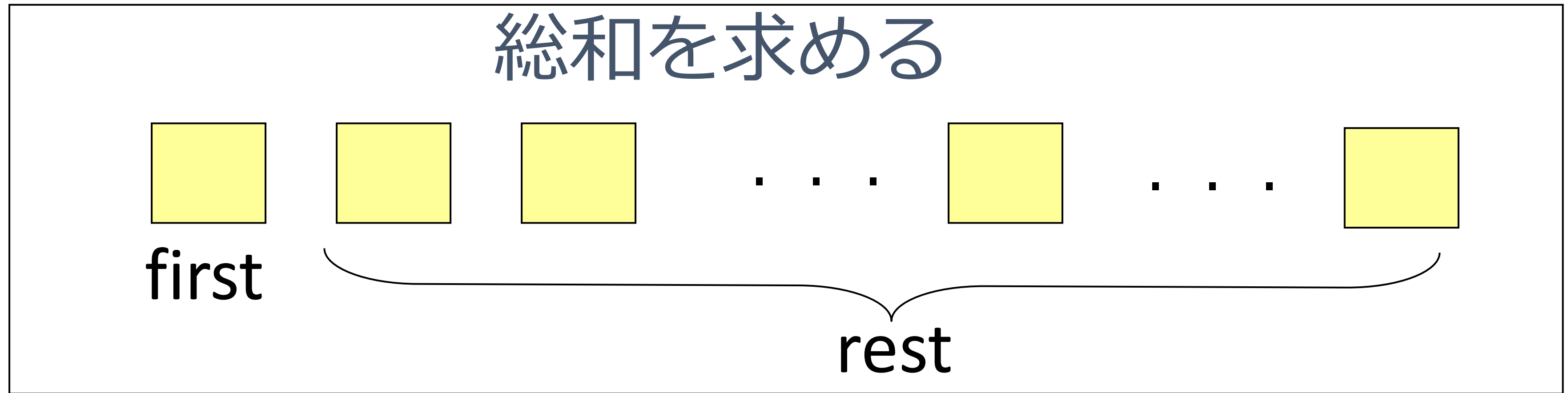
1. リストが空ならば：　→　終了条件

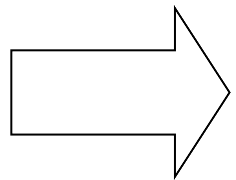
   0　→　自明な解

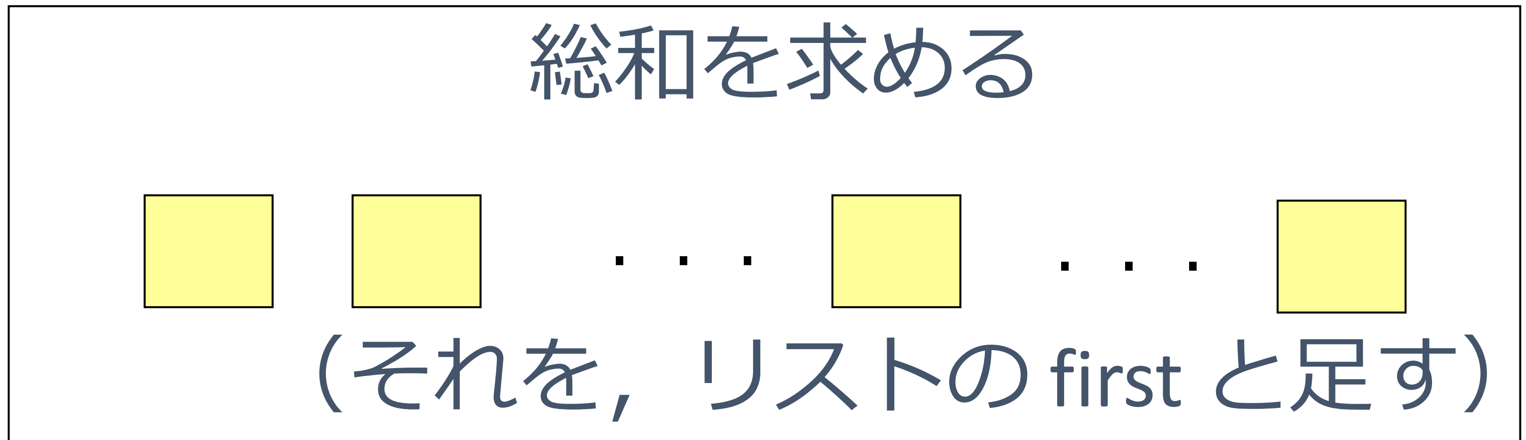
2. そうで無ければ：
   - リストの rest を求める（これもリスト）．「その総和と，リストの先頭との和」が，求める総和である

リストの総和

リストが空で無いとき





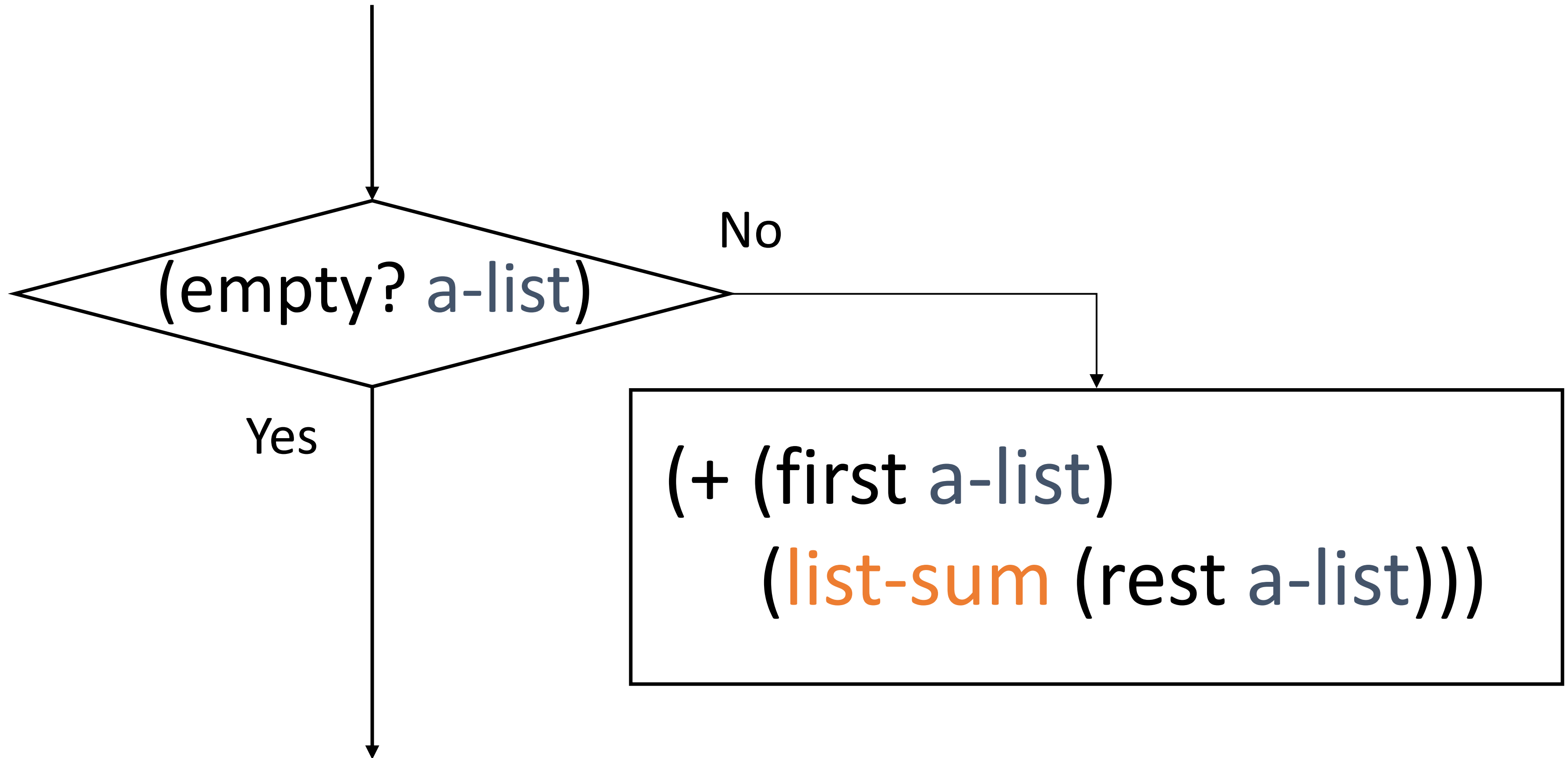


リストの総和

```
(define (list-sum a-list)
  (cond
    [(empty? a-list) 0]
    [else (+ (first a-list)
             (list-sum (rest a-
list))))]))
```

終了条件

自明な解



0 が自明な解である

リストの総和 list-sum

- list-sum の内部に list-sum が登場

```
(define (list-sum a-list)
  (cond
    [(empty? a-list) 0]
    [else (+ (first a-list)
             (list-sum (rest a-list)))]))
```

- sum の実行が繰り返される

例： (list-sum (list 1 2 3))
= (+ 1 (list-sum (list 2 3)))

例題２．ステップ実行

- 関数 list-sum （例題１）について，実行結果に至る過程を見る
  - (list-sum (list 1 2 3)) から 6 に至る過程を見る
  - DrScheme の stepper を使用する

```
= (list-sum (list 1 2 3))
= ...
= (+ 1 (list-sum (list 2 3)))
= ...
= (+ 1 (+ 2 (list-sum (list 3))))
= ...
= (+ 1 (+ 2 (+ 3 (list-sum empty))))
= ...
= (+ 1 (+ 2 (+ 3 0)))
= (+ 1 (+ 2 3))
= (+ 1 5)
= 6
```

基本的な計算式への展開（= (list-sum (list 1 2 3)) から = (+ 1 (+ 2 (+ 3 (list-sum empty)))) の次の = ... まで）

演算の実行（= (+ 1 (+ 2 (+ 3 0))) から = 6 まで）

「例題２．ステップ実行」の手順

Database Lab.

1. 次を「定義用ウインドウ」で，実行しなさい
   - Intermediate Student で実行すること
   - 入力した後に，Execute ボタンを押す

```
(define (list-sum a-list)
  (cond
    [(empty? a-list) 0]
    [else (+ (first a-list)
             (list-sum (rest a-list)))]))
(list-sum (list 1 2 3))
```

← 例題１と同じ

2. DrScheme を使って，ステップ実行の様子を確認しなさい　（Step ボタン，Next ボタンを使用）
   - 理解しながら進むこと

☆　次は，例題３に進んでください

(list-sum (list 1 2 3)) から 6 が得られる過程の概略

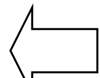
= (list-sum (list 1 2 3)) 最初の式
= ...
= (+ 1 (list-sum (list 2 3)))
= ...
= (+ 1 (+ 2 (list-sum (list 3))))
= ...
= (+ 1 (+ 2 (+ 3 (list-sum empty))))
= ...
= (+ 1 (+ 2 (+ 3 0)))
= (+ 1 (+ 2 3))
= (+ 1 5)
= 6 実行結果

コンピュータ内部での計算

(list 1 2 3) の和
⇩
(list 2 3) の和に1を足す
⇩
(list 3) の和に1,2を足す
⇩
emptyの和に1,2,3を足す

(list-sum (list 1 2 3)) から (+ 1 (list 2 3)) が得られる過程

```
= (list-sum (list 1 2 3))
= ...
= (+ 1 (list-sum (list 2 3)))
= ...
= (+ 1 (+ 2 (list-sum (list 3))))
= ...
= (+ 1 (+ 2 (+ 3 (list-sum em
= ...
= (+ 1 (+ 2 (+ 3 0)))
= (+ 1 (+ 2 3))
= (+ 1 5)
= 6
```

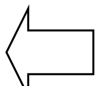


この部分は

```
(list-sum (list 1 2 3))
= (cond
    [(empty? (list 1 2 3)) 0]
    [else (+ (first (list 1 2 3))
             (list-sum (rest (list 1 2 3))))])
= (cond
    [false 0]
    [else (+ (first (list 1 2 3))
             (list-sum (rest (list 1 2 3))))])
= (+ (first (list 1 2 3))
     (list-sum (rest (list 1 2 3))))
= (+ 1
     (list-sum (rest (list 1 2 3))))
= (+ 1
     (list-sum (list 2 3)))
```

(list-sum (list 1 2 3)) から (+ 1 (list 2 3)) が得られる過程

```
= (list-sum (list 1 2 3))
= ...
= (+ 1 (list-sum (list 2 3)))
= ...
= (+ 1 (+ 2 (list-sum (list 3))))
= ...
= (
= ..
= (
= (
= (
= 6
```



この部分は

```
(list-sum (list 1 2 3))
= (cond
    [(empty? (list 1 2 3)) 0]
    [else (+ (first (list 1 2 3))
             (list-sum (rest (list 1 2 3))))])
= (cond
    [false 0]
    [else (+ (first (list 1 2 3))
             (list-sum (rest (list 1 2 3))))])
```

これは、

```
(define (sum a-list)
  (cond
    [(empty? a-list) 0]
    [else (+ (first a-list)
             (sum (rest a-list)))]))
```

の a-list を (list 1 2 3) で置き換えたもの

(contains-5? (list 3 5 7 9)) から (contains-5? (list 5 7 9))が得られる過程

(contains-5? (list 3 5 7 9))

= …

=(contains-5? (list 5 7 9))

= …

=

この部分は

```
(contains-5? (list 3 5 7 9))
= (cond
   [(empty? (list 3 5 7 9)) false]
   [else (cond
     [(= (first (list 3 5 7 9)) 5)  true]
     [else (contains-5? (rest (list 3 5 7 9)))])])
= (cond
   [false false]
   [else (cond
     [(= (first (list 3 5 7 9)) 5)  true]
```

これは,

```
(define (contains-5? a-list)
  (cond
    [(empty? a-list) false]
    [else (cond
      [(= (first a-list) 5)  true]
      [else (contains-5? (rest a-list))])]))
```

の a-list を (list 3 5 7 9) で置き換えたもの

例題３．平均点

- 点数のリストから，平均点を求めるプログラム average を作り，実行する
  - 点数のデータはリストとして扱う
  - 合計を求める関数 list-sum と，リストの長さを求める関数 length を組み合わせる

  list-sum, length については、以前の授　業の資料を参照のこと

平均点

平均点

＝　リストの総和／リストの長さ

「例題３．平均点」の手順

1. 次を「定義用ウインドウ」で，実行しなさい
   - 入力した後に，Execute ボタンを押す

```
;; list-sum: list -> number
;; total of a list
;; (list-sum (list 40 90 80)) = 210
(define (list-sum a-list)
  (cond
    [(empty? a-list) 0]
    [else (+ (first a-list)
             (list-sum (rest a-list)))]))
;; average: list -> number
;; average of a list
;; (average (list 40 90 80)) = 70
(define (average a-list)
  (/ (list-sum a-list) (length a-list)))
```

2. その後，次を「実行用ウインドウ」で実行しなさい

```
(average (list 40 90 80))
(average (list 100 200 300 400 500))
```

☆　次は，例題４に進んでください

Untitled - DrScheme

File Edit Windows Show Language Scheme Help

Untitled (define ...) Save Check Syntax Step Execute Break

```
;; list-sum: list -> number
;; total of a list
;; (list-sum (list 40 90 80)) = 210
(define (list-sum a-list)
  (cond
    [(empty? a-list) 0]
    [else (+ (first a-list)
             (list-sum (rest a-list)))]))
;; how-many: list -> number
;; length of a list
;; (how-many (list 40 90 80)) = 3
(define (how-many a-list)
  (cond
    [(empty? a-list) 0]
    [else (+ 1
             (how-many (rest a-list)))]))
;; average: list -> number
;; average of a list
;; (average (list 40 90 80)) = 70
(define (average a-list)
  (/ (list-sum a-list) (how-many a-list)))
```

```
>
```

まず，Scheme のプログラムを
コンピュータに読み込ませている

Database Lab.

File Edit Windows Show Language Scheme Help

Untitled (define ...) Save Check Syntax Step Execute Break

```
;; list-sum: list -> num
;; total of a list
;; (list-sum (list 40 90
(define (list-sum a-list
  (cond
    [(empty? a-list) 0]
    [else (+ (first a-li
             (list-sum (
;; how-many: list -> num
;; length of a list
;; (how-many (list 40 90
(define (how-many a-list
  (cond
    [(empty? a-list) 0]
    [else (+ 1
             (how-many (rest a-list)))]))
;; average: list -> number
;; average of a list
;; (average (list 40 90 80)) = 70
(define (average a-list)
  (/ (list-sum a-list) (how-many a-list)))
```

```
> (average (list 40 90 80))
70
> (average (list 100 200 300 400 500))
300
> (list-sum (list 40 90
210
> (how-many (list 40 90
3
>
```

これは，

(average (list 40 90 80))

と書いて，a-list の値を

(list 40 90 80) に設定しての実行

実行結果である「70」が表示される

Database Lab.

入力と出力

入力はリスト

出力は数値

平均点のプログラム

```
;; list-sum: list -> number
;; total of a list
;; (list-sum (list 40 90 80)) = 210
(define (list-sum a-list)
   (cond
     [(empty? a-list) 0]
     [else (+ (first a-list)
                  (list-sum (rest a-list)))]))
;; average: list -> number
;; average of a list
;; (average (list 40 90 80)) = 70
(define (average a-list)
    (/ (list-sum a-list) (length a-list)))
```

list-sum の部分

average の部分

list-sum, average の関係

- list-sum
  - 「数のリスト」から「リストの総和」を求める
- average
  - 「数のリスト」から「平均」を求める
  - list-sum を使用

例題４．ステップ実行

- 関数 average （例題３）について，実行結果に至る過程を見る
  - (average (list 40 90 80)) から 70 に至る過程を見る
  - DrScheme の stepper を使用する

```
(average (list 40 90 80))
= (/ (list-sum (list 40 90 80)) (length (list 40 90 80)))
= ...
= (/ 210 (length (list 40 90 80)))
= ...
= (/ 210 3)
= 70
```

Database Lab.

「例題４．ステップ実行」の手順

Database Lab.

1. 次を「定義用ウインドウ」で，実行しなさい
   - Intermediate Student で実行すること
   - 入力した後に，Execute ボタンを押す

```
;; list-sum: list -> number
;; total of a list
;; (list-sum (list 40 90 80)) = 210
(define (list-sum a-list)
  (cond
    [(empty? a-list) 0]
    [else (+ (first a-list)
             (list-sum (rest a-list)))]))
;; average: list -> number
;; average of a list
;; (average (list 40 90 80)) = 70
(define (average a-list)
  (/ (list-sum a-list) (length a-list)))
(list-sum (list 40 90 80))
```

例題３と同じ

2. DrScheme を使って，ステップ実行の様子を確認しなさい　（Step ボタン，Next ボタンを使用）
   - 理解しながら進むこと

☆　次は，例題５に進んでください

(average (list 40 90 80)) から 70 が得られる過程の概略

Database Lab.

(average (list 40 90 80)) 最初の式

= (/ (list-sum (list 40 90 80)) (length (list 40 90 80)))

= …

= (/ 210 (length (list 40 90 80)))

= ...

= (/ 210 3)

コンピュータ内部での計算

= 70 実行結果

(average (list 40 90 80)) から 70 が得られる過程の概略

(average (list 40 90 80))

= (/ (list-sum (list 40 90 80)) (length (list 40 90 80)))

= …

=

=

=

=

これは,

```
(define (average a-list)
    (/ (list-sum a-list) (length a-list)))
```

の a-list を (list 40 90 80) で置き換えたもの

例題５．「5」を含むか調べる

- リストの要素の中に「5」を含むかどうか調べる関数 contains-5? を作り，実行する

「例題５．「5」を含むか調べる」の手順

1. 次を「定義用ウインドウ」で，実行しなさい
   - 入力した後に，Execute ボタンを押す

```
;; contains-5?: list -> true or false
;; it investigates whether 5 is included
;; (contains-5? (list 3 5 7 9)) = true
(define (contains-5? a-list)
  (cond
    [(empty? a-list) false]
    [else (cond
        [(= (first a-list) 5)  true]
        [else (contains-5? (rest a-list))])]))
```

2. その後，次を「実行用ウインドウ」で実行しなさい

```
(contains-5? (list 1 2 3 4))
(contains-5? (list 3 5 7 9))
```

☆　次は，例題６に進んでください

Untitled - DrScheme

File Edit Windows Show Language Scheme Help

Untitled (define ...) Save Check Syntax Step Execute Break

```
;; contains-5?: list -> true or false
;; it investigates whether 5 is included
;; (contains-5? (list 3 5 7 9)) = true
(define (contains-5? a-list)
  (cond
    [(empty? a-list) false]
    [else (cond
            [(= (first a-list) 5)  true]
            [else (contains-5? (rest a-list))])]))
```

\>

3:3 Unlocked not running

まず，Scheme のプログラムを
コンピュータに読み込ませている

Untitled - DrScheme

File Edit Windows Show Language Scheme Help

Untitled (define ...) Save Check Syntax Step Execute Break

```
;; contains-5?: 
;; it investigat
;; (contains-5?
(define (contai
  (cond
    [(empty? a-l
    [else (cond
            [(=
            [else (contains-5? (rest a-list))])]))
```

これは,
(contains-5? (list 3 5 7 9))
と書いて，a-list の値を
(list 3 5 7 9) に設定しての実行

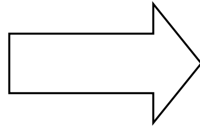
```
> (contains-5? (list 1 2 3 4))
false
> (contains-5? (list 3 5 7 9))
true
>
```

実行結果である「true」が
表示される

7:3 Unlocked not running

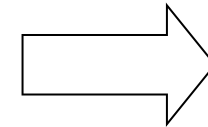
入力と出力

(list 3 5 7 9)



contains-5?

true



入力は
１つのリスト

出力は
true/false 値

contains-5? 関数

```
;; contains-5?: list -> true or false
;; it investigates whether 5 is included
;; (contains-5? (list 3 5 7 9)) = true
(define (contains-5? a-list)
  (cond
    [(empty? a-list) false]
    [else (cond
        [(= (first a-list) 5)  true]
        [else (contains-5? (rest a-list))])]))
```

「5」を含むか調べる

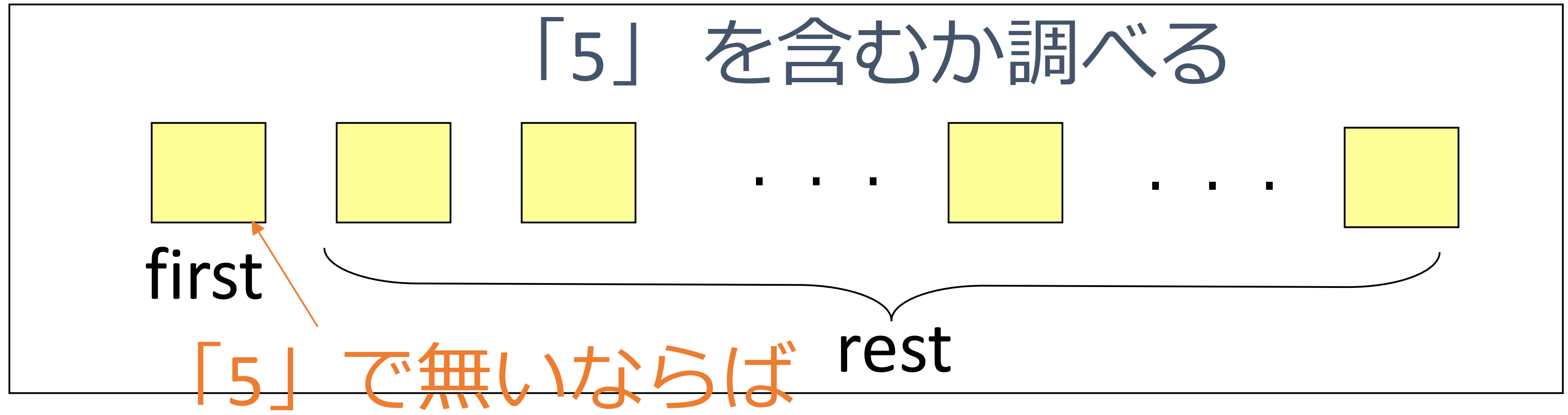
1. リストが空ならば：　→　終了条件

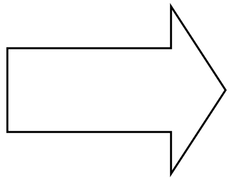
   false　→　自明な解

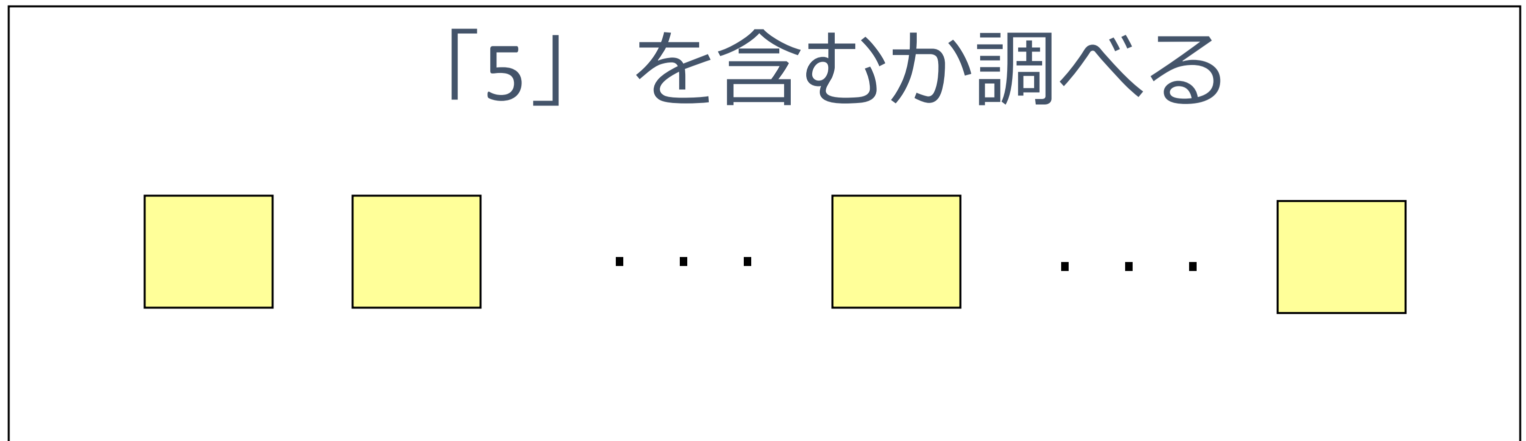
2. そうで無ければ：
   - リストの first が「5」であるかを調べる.
     - 「5」ならば： true
     - 「5」で無いならば: リストの rest が「5」を含むかどうかを調べる

「5」を含むか調べる

リストが空で無いとき





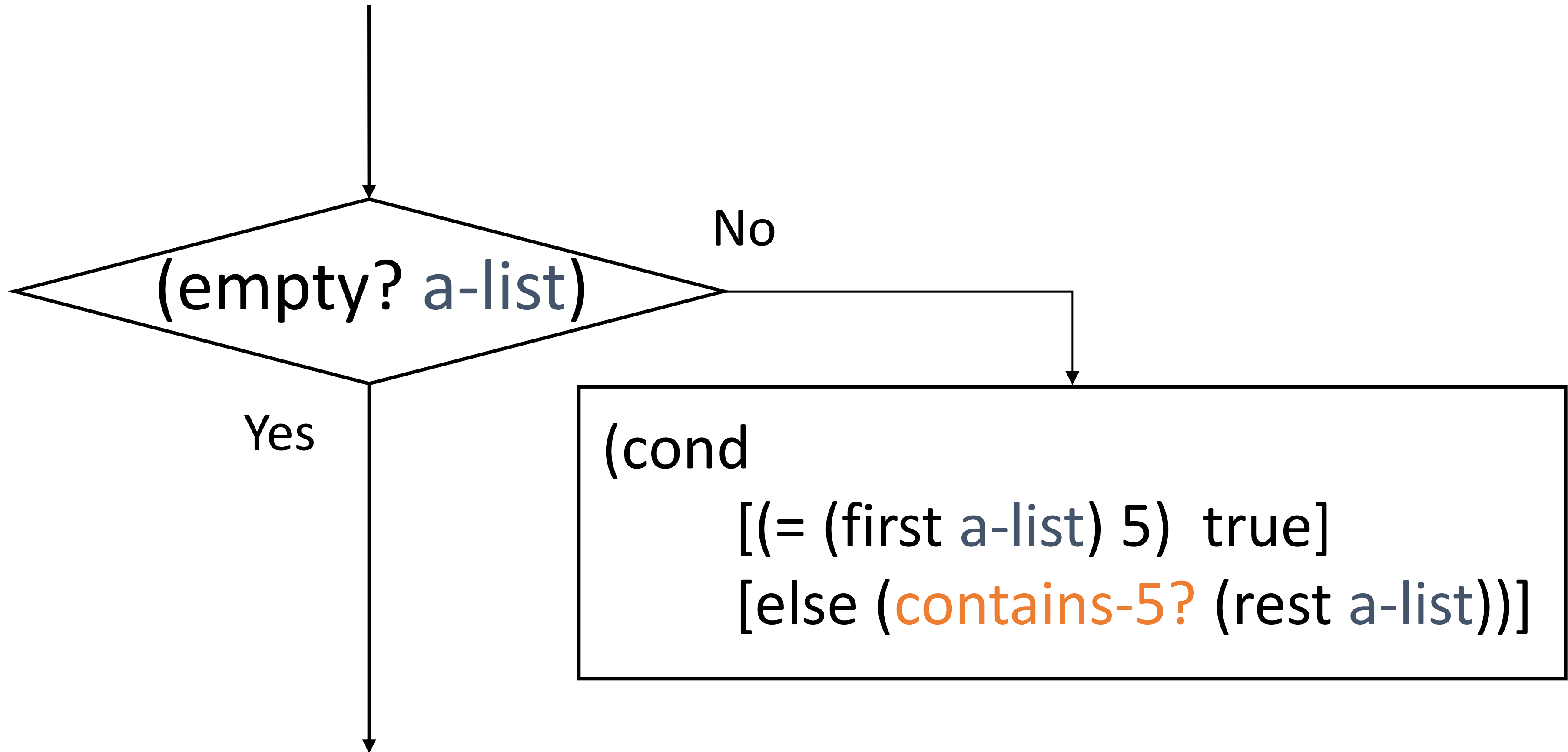


「5」を含むか調べる

```
;; contains-5?: list -> true or false
;; it investigates whether 5 is included
;; (contains-5? (list 3 5 7 9)) = true
(define (contains-5? a-list)
  (cond
    [(empty? a-list) false]
    [else (cond
      [(= (first a-list) 5)  true]
      [else (contains-5? (rest a-list))])]))
```

終了条件：(empty? a-list)

自明な解：false



false が自明な解である

「5」を含むか調べる contains-5?

- contains-5? の内部に contains-5? が登場

```
(define (contains-5? a-list)
  (cond
    [(empty? a-list) false]
    [else (cond
        [(= (first a-list) 5)  true]
        [else (contains-5? (rest a-list))])]))
```

- contains-5? の実行が繰り返される

例： (contains-5? (list 3 5 7 9))
　　= (contains-5? (list 5 7 9))

例題６．ステップ実行

Database Lab.

- 関数 contains-5? （例題５）について，実行結果に至る過程を見る
  - (contains-5? (list 3 5 7 9)) から true に至る過程を見る
  - DrScheme の stepper を使用する

```
(contains-5? (list 3 5 7 9))
= ...
=(contains-5? (list 5 7 9))
= ...
= true
```

「例題６．ステップ実行」の手順

Database Lab.

1. 次を「定義用ウインドウ」で，実行しなさい
   - Intermediate Student で実行すること
   - 入力した後に，Execute ボタンを押す

```
(define (contains-5? a-list)
  (cond
    [(empty? a-list) false]
    [else (cond
       [(= (first a-list) 5)  true]
       [else (contains-5? (rest a-list))])]))
(contains-5? (list 3 5 7 9))
```

例題５と同じ

2. DrScheme を使って，ステップ実行の様子を確認しなさい　（Step ボタン，Next ボタンを使用）
   - 理解しながら進むこと

☆　次は，例題７に進んでください

(contains-5? (list 3 5 7 9)) から true が得られる過程の概略

(contains-5? (list 3 5 7 9)) 最初の式

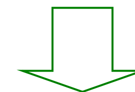
= …

=(contains-5? (list 5 7 9))

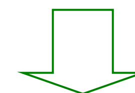
= … コンピュータ内部での計算

= true 実行結果

(list 3 5 7 9) から探す



(list 5 7 9) から探す



先頭が 5 なので

true

(contains-5? (list 3 5 7 9)) から (contains-5? (list 5 7 9))が得られる過程

(contains-5? (list 3 5 7 9))
= …
=(contains-5? (list 5 7 9))
= …
= true

この部分は

```
(contains-5? (list 3 5 7 9))
= (cond
    [(empty? (list 3 5 7 9)) false]
    [else (cond
       [(= (first (list 3 5 7 9)) 5)  true]
       [else (contains-5? (rest (list 3 5 7 9)))])])
= (cond
    [false false]
    [else (cond
       [(= (first (list 3 5 7 9)) 5)  true]
       [else (contains-5? (rest (list 3 5 7 9)))])])
= (cond
       [(= (first (list 3 5 7 9)) 5)  true]
       [else (contains-5? (rest (list 3 5 7 9)))])
= (cond
       [(= 3 5)  true]
       [else (contains-5? (rest (list 3 5 7 9)))])
= (cond
       [false  true]
       [else (contains-5? (rest (list 3 5 7 9)))])
= (contains-5? (rest (list 3 5 7 9)))
= (contains-5? (list 5 7 9))
```

(contains-5? (list 3 5 7 9)) から (contains-5? (list 5 7 9))が得られる過程

(contains-5? (list 3 5 7 9))
= …
=(contains-5? (list 5 7 9))
= …
=

この部分は

```
(contains-5? (list 3 5 7 9))
= (cond
   [(empty? (list 3 5 7 9)) false]
   [else (cond
      [(= (first (list 3 5 7 9)) 5)  true]
      [else (contains-5? (rest (list 3 5 7 9)))])])
= (cond
   [false false]
   [else (cond
      [(= (first (list 3 5 7 9)) 5)  true]
```

これは,

```
(define (contains-5? a-list)
  (cond
    [(empty? a-list) false]
    [else (cond
      [(= (first a-list) 5)  true]
      [else (contains-5? (rest a-list))])]))
```

の a-list を (list 3 5 7 9) で置き換えたもの

例題７．ベクトルの内積

- ２つのベクトルデータ x, y から２つのベクトルの内積を求めるプログラム product を作り，実行する
  - ２つのベクトルデータ x, y はリストとして扱う

「例題７．ベクトルの内積」の手順

1. 次を「定義用ウインドウ」で，実行しなさい
   - 入力した後に，Execute ボタンを押す

```
;; product: list list -> number
;; inner product of two vectors
;; (product (list 1 2 3) (list 4 5 6)) = 32
(define (product x y)
  (cond
    [(empty? x) 0]
    [else (+ (* (first x) (first y))
             (product (rest x) (rest y)))]))
```

2. その後，次を「実行用ウインドウ」で実行しなさい

```
(product (list 1 2 3) (list 4 5 6))
```

☆　次は，例題８に進んでください

Untitled - DrScheme

File Edit Windows Show Language Scheme Help

Untitled (define ...) Save Check Syntax Step Execute Break

```
;; product: list list -> number
;; inner product of two vectors
;; (product (list 1 2 3) (list 4 5 6)) = 32
(define (product x y)
  (cond
    [(empty? x) 0]
    [else (+ (* (first x) (first y))
             (product (rest x) (rest y)))]))
```

```
>
```

まず，Scheme のプログラムを
コンピュータに読み込ませている

3:3 Unlocked not running

Untitled - DrScheme

File Edit Windows Show Language Scheme Help

Untitled (define ...) Save Check Syntax Step Execute Break

```
;; produ
;; inner
;; (prod
(define
  (cond
    [(em
    [els
          (product (rest x) (rest y)))]))
```

これは,

(product (list 1 2 3) (list 4 5 6))

と書いて, x の値を(list 1 2 3) に,
y の値を (list 4 5 6) に設定しての実行

```
> (product (list 1 2 3) (list 4 5 6))
32
> (product (list 100 200) (list 200 300))
80000
>
```

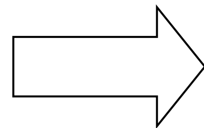
実行結果である「32」が
表示される

7:3 Unlocked not running

入力と出力

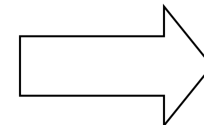
(list 1 2 3)

(list 4 5 6)



入力

32



出力

入力は，
２つのリスト

出力は数値

product 関数

```
;; product: list list -> number
;; inner product of two vectors
;; (product (list 1 2 3) (list 4 5 6)) = 32
(define (product x y)
  (cond
    [(empty? x) 0]
    [else (+ (* (first x) (first y))
             (product (rest x) (rest y)))]))
```

よくある勘違い

```
(define (product x y)
   (cond
      [(= x empty) 0]
      [else (+ (* (first x) (first y))
                  (product (rest x) (rest y)))]))
```

終了条件の判定：
- 正しくは「(empty? x)」
- x がリストのとき、(= x empty) はエラー
- 「=」は数値の比較には使えるが，リスト同士の比較には**使えない**

ベクトルの内積

1. リストが空ならば：　→　終了条件

   0　→　自明な解

2. そうで無ければ：

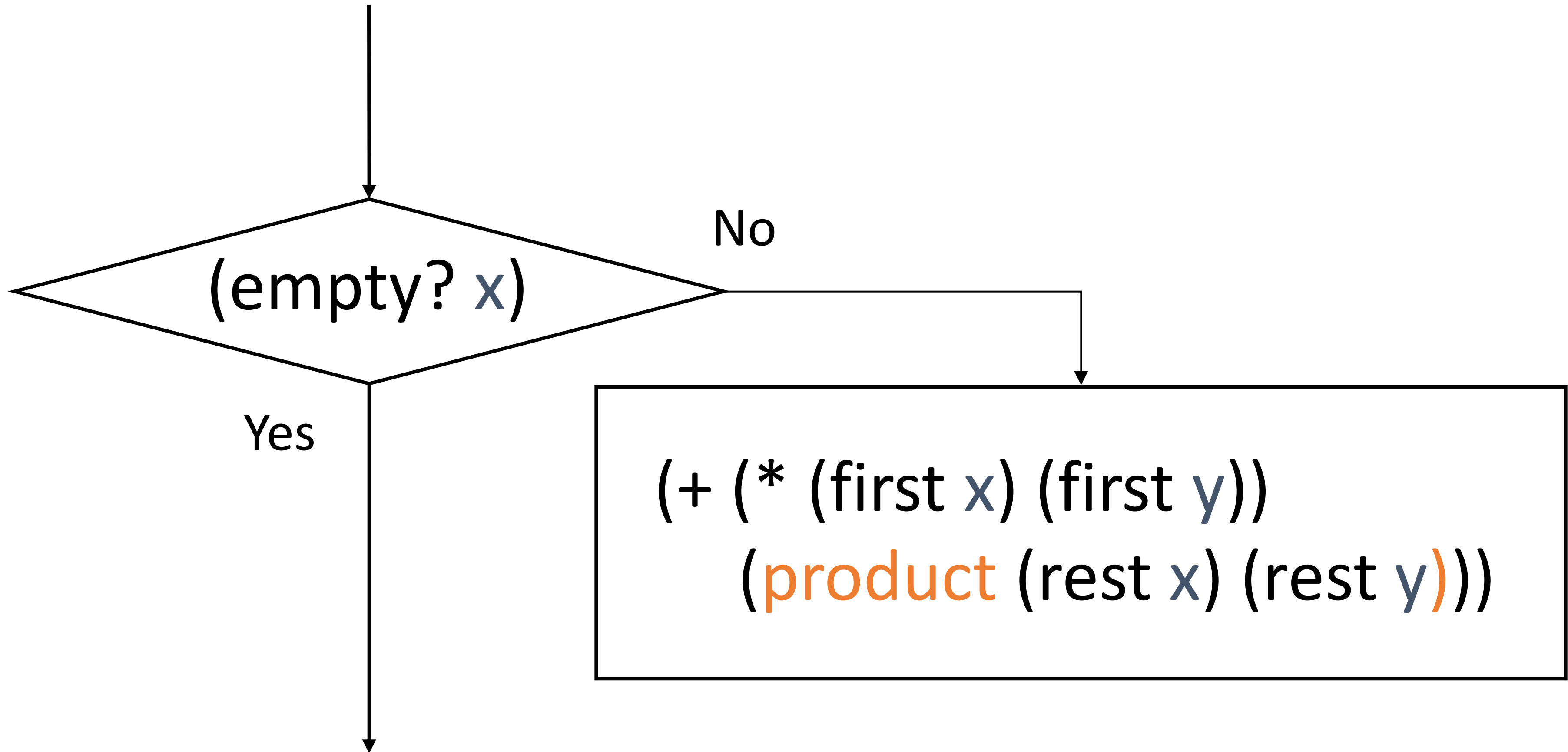
   – 「２つのリストの rest の内積と，２つのリストの先頭の積との和」　が求める解

ベクトルの内積

```
;; product: list list -> number
;; inner product of two vectors
;; (product (list 1 2 3) (list 4 5 6)) = 32
(define (product x y)
  (cond
    [(empty? x) 0]
    [else (+ (* (first x) (first y))
             (product (rest x) (rest y)))]))
```

終了条件: [(empty? x)

自明な解: 0


No
(empty? x)
Yes
(+ (* (first x) (first y))
(product (rest x) (rest y)))
0 が自明な解である

ベクトルの内積 product

- product の内部に product が登場

```
(define (product x y)
  (cond
    [(empty? x) 0]
    [else (+ (* (first x) (first y))
             (product (rest x) (rest y)))]))
```

- product の実行が繰り返される

例： (product (list 1 2 3) (list 4 5 6))

= (+ (* 1 4) (product (list 2 3) (list 5 6)))

例題８．ステップ実行

- 関数 product （例題７）について，実行結果に至る過程を見る
  - (product (list 1 2 3) (list 4 5 6)) から 32 に至る過程を見る
  - DrScheme の stepper を使用する

```
(product (list 1 2 3) (list 4 5 6))
= ...
= (+ (* 1 4)
     (product (list 2 3) (list 5 6)))
= ...
= (+ 4
     (+ 10
        (product (list 3) (list 6))))
= ...
= (+ 4 (+ 10 (+ 18 (product empty empty))))
= ...
= 32
```

基本的な計算式への展開

演算の実行

「例題８．ステップ実行」の手順

Database Lab.

1. 次を「定義用ウインドウ」で，実行しなさい
   - Intermediate Student で実行すること
   - 入力した後に，Execute ボタンを押す

```
(define (product x y)
   (cond
      [(empty? x) 0]
      [else (+ (* (first x) (first y))
                  (product (rest x) (rest y)))]))
(product (list 1 2 3) (list 4 5 6))
```

例題７と同じ

2. DrScheme を使って，ステップ実行の様子を確認しなさい　（Step ボタン，Next ボタンを使用）
   - 理解しながら進むこと

☆　次は，課題に進んでください

(product (list 1 2 3) (list 4 5 6)) から 32 が得られる過程の概略

(product (list 1 2 3) (list 4 5 6))

最初の式

```
= …
= (+ (* 1 4)
      (product (list 2 3) (list 5 6)))
= …
= (+ 4
      (+ 10
         (product (list 3) (list 6))))
= …
= (+ 4 (+ 10 (+ 18 (product empty empty))))
= …
= 32
```

コンピュータ内部での計算

実行結果

(product (list 1 2 3) (list 4 5 6)) から (+ 4 (product (list 2 3) (list 5 6))) が得られる過程

```
(product (list 1 2 3) (list 4 5 6))
= …
= (+ 4
    (product (list 2 3) (list 5 6))
= …
= (+ 4
    (+ 10
      (product (list 3) (list 6))))
= …
= (+ 4 (+ 10 (+ 18 (product en
= …
= 32
```

この部分は

```
(product (list 1 2 3) (list 4 5 6))
=  (cond
     [(empty? (list 1 2 3)) 0]
     [else (+ (* (first (list 1 2 3)) (first (list 4 5 6)))
              (product (rest (list 1 2 3)) (rest (list 4 5 6))))])
=  (cond
     [false 0]
     [else (+ (* (first (list 1 2 3)) (first (list 4 5 6)))
              (product (rest (list 1 2 3)) (rest (list 4 5 6))))])
= (+ (* (first (list 1 2 3)) (first (list 4 5 6)))
              (product (rest (list 1 2 3)) (rest (list 4 5 6))))
= (+ (* 1 (first (list 4 5 6)))
              (product (rest (list 1 2 3)) (rest (list 4 5 6))))
= (+ (* 1 4)
              (product (rest (list 1 2 3)) (rest (list 4 5 6))))
= (+ 4 (product (rest (list 1 2 3)) (rest (list 4 5 6))))
= (+ 4 (product (list 2 3) (rest (list 4 5 6))))
= (+ 4 (product (list 2 3) (list 5 6)))
```

(product (list 1 2 3) (list 4 5 6)) から (+ 4 (product (list 2 3) (list 5 6))) が得られる過程

```
(product (list 1 2 3) (list 4 5 6))
= …
= (+ 4
    (product (list 2 3) (list 5 6)))
= …
= …
= 32
```

この部分は

```
(product (list 1 2 3) (list 4 5 6))
= (cond
    [(empty? (list 1 2 3)) 0]
    [else (+ (* (first (list 1 2 3)) (first (list 4 5 6)))
             (product (rest (list 1 2 3)) (rest (list 4 5 6))))])
= (cond
    [false 0]
    [else (+ (* (first (list 1 2 3)) (first (list 4 5 6)))
             (product (rest (list 1 2 3)) (rest (list 4 5 6))))])
= (+ (* (first (list 1 2 3)) (first (list 4 5 6)))
```

これは,

```
(define (product x y)
  (cond
    [(empty? x) 0]
    [else (+ (* (first x) (first y))
             (product (rest x) (rest y)))]))
```

の x を (list 1 2 3) で， y を (list 4 5 6) で置き換えたもの

今日のパソコン演習課題

Database Lab.

課題 1

- 関数 list-sum （授業の例題 1 ）についての問題
  - (list-sum (list 1 2 3)) から 6 が得られる過程の概略を数行程度で説明しなさい
  - DrScheme の stepper を使うと，すぐに分かる

```
(define (list-sum a-list)
  (cond
    [(empty? a-list) 0]
    [else (+ (first a-list)
             (list-sum (rest a-list)))]))
```

課題２

DrScheme の stepper を利用した実行エラーの解決に関する問題

1. まずは，次を「定義用ウインドウ」で，実行しなさい
   – 入力した後に，Execute ボタンを押す

     ⇒　すると，実行用ウインドウに（赤い文字で）エラーメッセージが表示される（これは実行エラー）

```
(define (list-length a-list)
  (cond
    [(empty? a-list) 0]
    [else (+ 1
             (list-length (rest a-list)))]))
(list-length 100)
```

2. DrScheme で stepper を使ってステップ実行を行って，エラーの箇所を特定しなさい．エラーの原因について報告しなさい．

課題３

シンボル出現の判定プログラム作成

- シンボルのリスト a-list と，シンボル a-symbol から，a-list が a-symbol を含むときに限り true を返す関数 contains? を作りなさい

課題３のヒント：
すべての要素が１０以上か？

- リストの要素を調べ,
  - すべての要素が１０以上　→　true
  - そうでなければ　→　false

```
(define (all-are-large alon)
  (cond
    [(empty? alon) true]
    [else (and (<= 10 (first alon))
               (all-are-large (rest alon)))]))
```

課題４

- リストの要素の中に 「偶数」を含むかどうか調べる関数を作りなさい
  - 偶数を１つでも含めば true.　１つも含まなければ false
  - even? を使うこと

課題 5

- n次の多項式

  $f(x) = a0 + a1・x + a2・x^{2} + ・・・ + an^{n}・x$

  について，次数 n と，係数 a0 から an から，f(x) を計算するプログラムを作りなさい
  - 次ページ以降で説明する Horner法を使うこと

多項式の計算

- n次の多項式

  $f(x) = a_0 + a_1 \cdot x + a_2 \cdot x^2 + \cdot\cdot\cdot + a_n \cdot x^n$

  について，次数 n，係数 a_0 から a_n と x から f(x) を計算する

Horner法による多項式の計算

$$f(x) = a_0 + a_1 \cdot x + a_2 \cdot x^2 + \cdots + a_n \cdot x^n$$
$$= a_0 + (a_1 + (a_2 + \cdots + (a_{n-1} + a_n \cdot x) x \cdots) x) x$$

例えば，　$5 + 6x + 3x^2$
$= 5 + (6 + 3x) x$

<u>計算手順</u>

① a_n

② $a_{n-1} + a_n \cdot x$

③ $a_{n-2} + (a_{n-1} + a_n \cdot x) x$

・・・　(a_0 まで続ける)

さらに勉強したい人への
補足説明資料

リストに関係する関数

- (length list)

  リストの要素の個数

- (list-ref list n)

  リストのｎ番目の要素（先頭は０番目）

これらの関数と同じ機能を持つ関数
my-length, my-list-ref を敢えて書いて
みた例を以下に紹介する

例題９．リストのｎ番目の要素

- リストのｎ番目の要素（先頭は０番目）を得る関数 my-list-ref を作り，実行する

Untitled - DrScheme

File Edit Windows Show Language Scheme Help

Untitled (define ...) Save Check Syntax Step Execute Break

```
(define (my-list-ref a-list n)
  (cond
    [(= n 0) (first a-list)]
    [else (my-list-ref (rest a-list) (- n 1))]))
```

```
>
```

まず，Scheme のプログラムを
コンピュータに読み込ませてい

3:3 Unlocked not running

Database Lab.

Untitled - DrScheme

File Edit Windows Show Language Scheme Help

Untitled (define ...) Save

```
(define (my-li
  (cond
    [(= n 0) (
    [else (my-
```

```
> (my-list-ref (list 11 12 13 14) 0)
11
> (my-list-ref (list 11 12 13 14) 1)
12
> (my-list-ref (list 11 12 13 14) 2)
13
>
```

9:3 Unlocked not running

これは，

(my-list-ref (list 11 12 13 14) 1)

と書いて，a-list の値を (list 11 12 13 14) に，n の値を 1 に設定しての実行

実行結果である「12」が表示される

入力と出力

my-list-ref 関数

「関数である」ことを
示すキーワード

関数の名前

```
(define (my-list-ref a-list n)
  (cond
    [(= n 0) (first a-list)]
    [else (my-list-ref (rest a-list) (- n 1))]))
```

値を2つ受け取る（入力）

a-list と n の値から
n 番目の要素を求める（出力）

(my-list-ref (list 11 12 13 14) 1) から12が得られる過程

Database Lab.

(my-list-ref (list 11 12 13 14) 1)

最初の式

```
= (cond
    [(= 1 0) (first (list 11 12 13 14))]
    [else (my-list-ref (rest (list 11 12 13 14) ) (- 1 1))])
= (cond
    [false (first (list 11 12 13 14))]
    [else (my-list-ref (rest (list 11 12 13 14) ) (- 1 1))])
= (my-list-ref (rest (list 11 12 13 14) ) (- 1 1))
= (my-list-ref (list 12 13 14) (- 1 1))
= (my-list-ref (list 12 13 14) 0)
=  (cond
    [(= 0 0) (first (list 12 13 14))]
    [else (my-list-ref (rest (list 12 13 14) ) (- 0 1))])
= (cond
    [true (first (list 12 13 14))]
    [else (my-list-ref (rest (list 12 13 14) ) (- 0 1))])
= (first (list 12 13 14))
= 12
```

コンピュータ内部での計算

実行結果

(my-list-ref (list 11 12 13 14) 1) から12が得られる過程

Database Lab.

```
(my-list-ref (list 11 12 13 14) 1)
= (cond
    [(= 1 0) (first (list 11 12 13 14))]
    [else (my-list-ref (rest (list 11 12 13 14) ) (- 1 1))])
= (cond
    [false (first (list 11 12 13 14))]
```

これは,

```
(define (my-list-ref a-list n)
  (cond
    [(= n 0) (first a-list)]
    [else (my-list-ref (rest a-list) (- n 1))]))
```

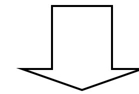
の a-list を (list 11 12 13 14) で， n を 1 で置き換えたもの

```
= (first (list 12 13 14))
= 12
```

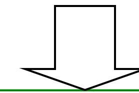
(my-list-ref (list 11 12 13 14) 1) から12が得られる過程の概略

(my-list-ref (list 11 12 13 14) 1)
= …
= (my-list-ref (list 12 13 14) 0)
= …
= (first (list 12 13 14))
= 12

リストの1番目の要素



リスト rest を取って，その0番目の要素



リストの先頭

Stepper

File Edit Windows Help

Home | << Previous | Next >>

```
(define (list-nth a-list n)
  (cond
   ((= n 1) (first a-list))
   (else (list-nth (rest a-list) (- n 1)))))
```

```
(list-nth
  (list 11 12 13 14)
  2)
```

```
(cond
 ((= 2 1)
  (first
   (list 11 12 13 14)))
 (else
  (list-nth
    (rest
     (list 11 12 13 14))
    (- 2 1))))
```

my-list-ref の「a-list」は「(list 11 12 13 14)」で
「n」は「1」で置き換わる

Stepper

File Edit Windows Help

Home | << Previous | Next >>

```
(define (list-nth a-list n)
  (cond
   ((= n 1) (first a-list))
   (else (list-nth (rest a-list) (- n 1)))))
```

```
(cond
 ((= 2 1)
  (first
   (list 11 12 13 14)))
 (else
  (list-nth
    (rest
     (list 11 12 13 14))
    (- 2 1))))
```

➡

```
(cond
 (false
  (first
   (list 11 12 13 14)))
 (else
  (list-nth
    (rest
     (list 11 12 13 14))
    (- 2 1))))
```

「(= 1 0)」は
「false」で置き換わる

Stepper

File Edit Windows Help

Home | << Previous | Next >>

```
(define (list-nth a-list n)
  (cond
   ((= n 1) (first a-list))
   (else (list-nth (rest a-list) (- n 1)))))
```

```
(cond
 (false
  (first
   (list 11 12 13 14)))
 (else
  (list-nth
    (rest
     (list 11 12 13 14))
    (- 2 1))))
```

→

```
(list-nth
  (rest
   (list 11 12 13 14))
  (- 2 1))
```

「(cond [false 式 X] [else 式 Y])」 は
「式 Y 」 で置き換わる

Stepper

File Edit Windows Help

Home | << Previous | Next >>

```
(define (list-nth a-list n)
  (cond
   ((= n 1) (first a-list))
   (else (list-nth (rest a-list) (- n 1)))))
```

```
(list-nth
  (rest
   (list 11 12 13 14))
  (- 2 1))
```

→

```
(list-nth
  (list 12 13 14)
  (- 2 1))
```

「(rest (list 11 12 13 14))」は
「(list 12 13 14)」で置き換わる

Stepper

File Edit Windows Help

Home | << Previous | Next >>

```
(define (list-nth a-list n)
  (cond
   ((= n 1) (first a-list))
   (else (list-nth (rest a-list) (- n 1)))))
```

```
(list-nth                      ➡  (list-nth
  (list 12 13 14)                    (list 12 13 14)
  (- 2 1))                           1)
```

「(- 1 1)」は「0」で置き換わる

Stepper

File Edit Windows Help

Home | << Previous | Next >>

```
(define (list-nth a-list n)
  (cond
   ((= n 1) (first a-list))
   (else (list-nth (rest a-list) (- n 1)))))
```

```
(list-nth
  (list 12 13 14)
  1)
```

→

```
(cond
 ((= 1 1)
  (first (list 12 13 14)))
 (else
  (list-nth
    (rest (list 12 13 14))
    (- 1 1))))
```

my-list-ref の「a-list」は「(list 12 13 14)」で，「n」は「0」で置き換わる

Stepper

File Edit Windows Help

Home | << Previous | Next >>

```
(define (list-nth a-list n)
  (cond
   ((= n 1) (first a-list))
   (else (list-nth (rest a-list) (- n 1)))))
```

```
(cond                                 (cond
 ((= 1 1)                              (true
  (first (list 12 13 14)))              (first (list 12 13 14)))
 (else                          ➡     (else
  (list-nth                             (list-nth
    (rest (list 12 13 14))                (rest (list 12 13 14))
    (- 1 1))))                            (- 1 1))))
```

「(= 0 0)」は
「true」で置き換わる

Stepper

File Edit Windows Help

Home | << Previous | Next >>

```
(define (list-nth a-list n)
  (cond
   ((= n 1) (first a-list))
   (else (list-nth (rest a-list) (- n 1)))))
```

```
(cond
 (true
  (first (list 12 13 14)))
 (else
  (list-nth
    (rest (list 12 13 14))
    (- 1 1))))
```

→

```
(first (list 12 13 14))
```

「(cond [true 式Ｘ] [else 式Ｙ])」は「式Ｘ」で置き換わる

Stepper

File Edit Windows Help

Home | << Previous | Next >>

```
(define (list-nth a-list n)
  (cond
   ((= n 1) (first a-list))
   (else (list-nth (rest a-list) (- n 1)))))
```

```
(first (list 12 13 14))        → 12
```

「(first (list 12 13 14))」は「12」で置き換わる

リストの n 番目の要素

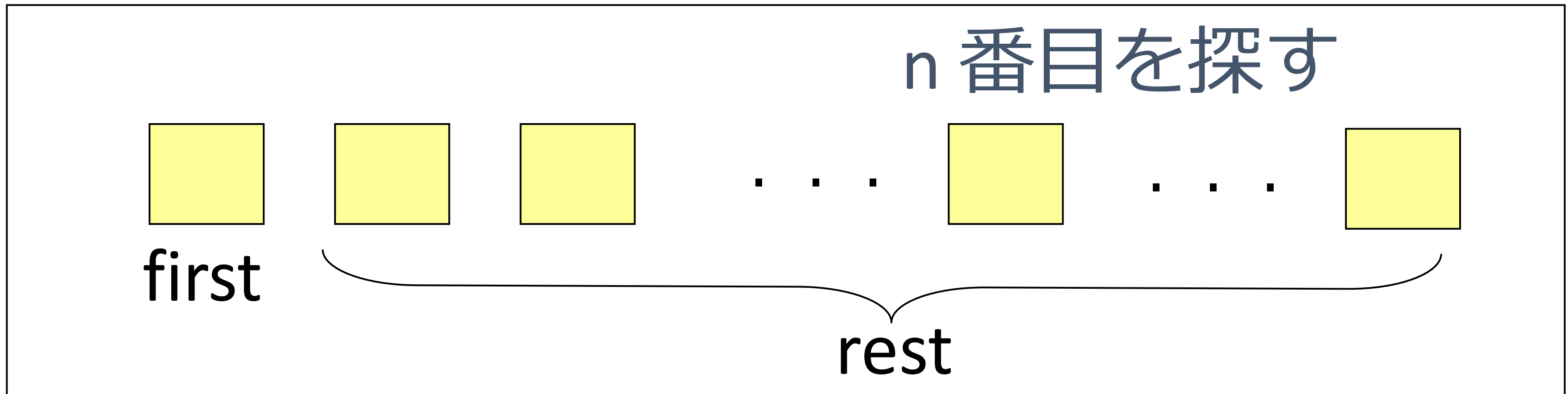
1. n = 0 ならば： → 終了条件

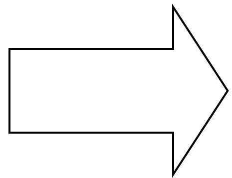
   リストの first → 自明な解

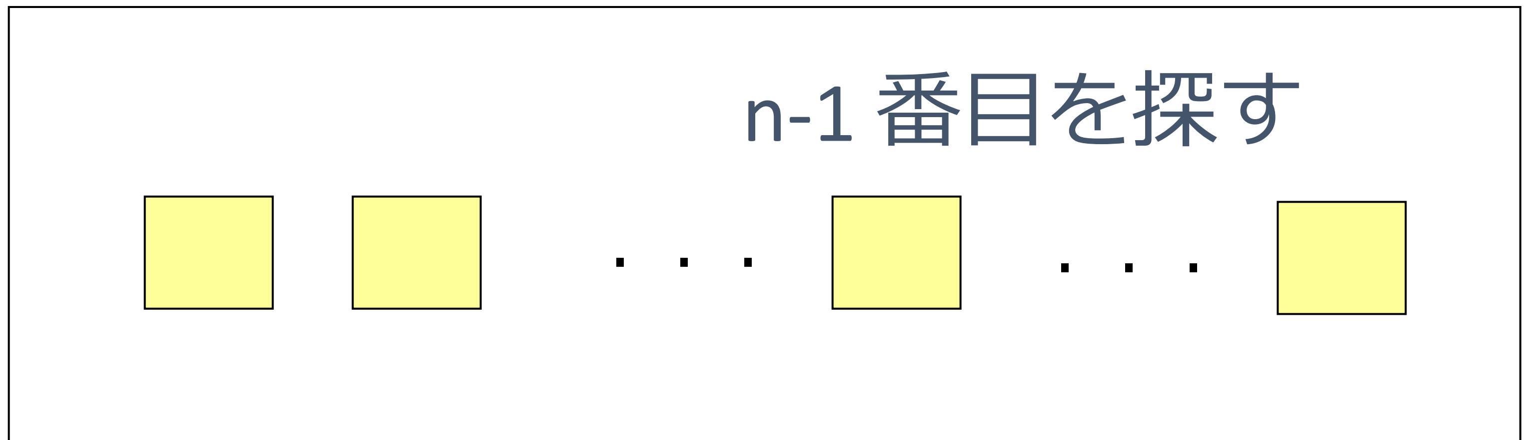
2. そうで無ければ：
   - 「リストの rest を求める（これもリスト）．その n-1 番目の要素」が求める解

リストの n 番目の要素

n > 0 のとき





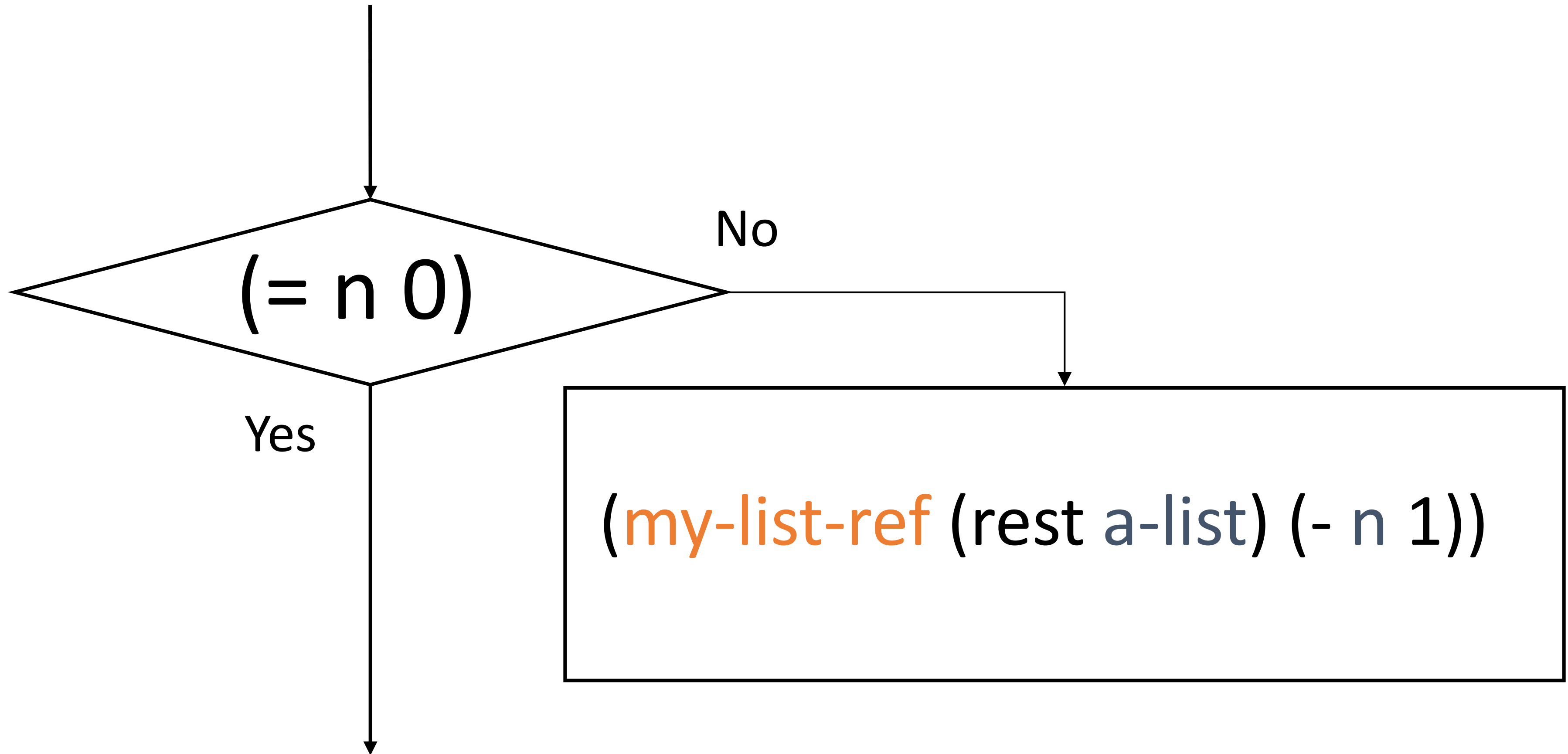


リストの n 番目の要素

```
(define (my-list-ref a-list n)
  (cond
    [(= n 0) (first a-list)]
    [else (my-list-ref (rest a-list) (- n 1))]))
```

終了条件: (= n 0)

自明な解: (first a-list)



(first a-list) が自明な解である

リストの n 番目の要素 my-list-ref

- my-list-ref の内部に my-list-ref が登場

```
(define (my-list-ref a-list n)
  (cond
    [(= n 0) (first a-list)]
    [else (my-list-ref (rest a-list) (- n 1))]))
```

- my-list-ref の実行が繰り返される

  例： (my-list-ref (list 11 12 13 14) 1)

  = (my-list-ref (list 12 13 14) 0)

例題１０．リストの長さ

- リストの要素の個数を求める関数 my-length を作り，実行する

Untitled - DrScheme

File Edit Windows Show Language Scheme Help

Untitled
(define ...)
Save
Check Syntax
Step
Execute
Break

```
(define (my-length a-list)
  (cond
    [(empty? a-list) 0]
    [else (+ 1
               (my-length (rest a-list)))]))
```

```
>
```

まず，Schemeのプログラムを
コンピュータに読み込ませている

3:3
Unlocked
not running

Untitled - DrScheme

File Edit Windows Show Language Scheme Help

Untitled (define ...) Save Check Syntax Step Execute Break

```
(define (my
  (cond
    [(empty
    [else (
```

これは，

(my-length (list 11 12))

と書いて，a-list の値を

(list 11 12) に設定しての実行

```
> (my-length (list 11 12))
2
> (my-length (list 100 200 300 400))
4
> (my-leng
0
>
```

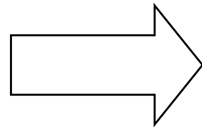
実行結果である「2」が

表示される

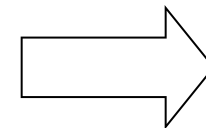
9:3 Unlocked not running

入力と出力

(list 11 12)



2



入力

出力

my-length 関数

「関数である」ことを
示すキーワード

関数の名前

```
(define (my-length a-list)
  (cond
    [(empty? a-list) 0]
    [else (+ 1
            (my-length (rest a-list)))]))
```

値を1つ受け取る（入力）

a-list の値から
リストの長さを求める（出力）

(my-list-ref (list 11 12)) から 2 が得られる過程 (1/2)

(my-length (list 11 12))　最初の式

```
= (cond
    [(empty? (list 11 12)) 0]
    [else (+ 1 (my-length (rest (list 11 12))))])
= (cond
    [false 0]
    [else (+ 1 (my-length (rest (list 11 12))))])
= (+ 1 (my-length (rest (list 11 12))))
= (+ 1 (my-length (list 12)))
= (+ 1 (cond
          [(empty? (list 12)) 0]
          [else (+ 1 (my-length (rest (list 12))))]))
= (+ 1 (cond
          [false 0]
          [else (+ 1 (my-length (rest (list 12))))]))
```

コンピュータ内部での計算

(my-list-ref (list 11 12)) から 2 が得られる過程 (2/2)

Database Lab.

```
= (+ 1 (+ 1 (my-length (rest (list 12)))))
= (+ 1 (+ 1 (my-length empty)))
= (+ 1 (+ 1 (cond
                [(empty? empty) 0]
                [else (+ 1 (my-length empty))])))
= (+ 1 (+ 1 (cond
                [true 0]
                [else (+ 1 (my-length empty))])))
= (+ 1 (+ 1 0))
= (+ 1 1)
= 2
```

コンピュータ内部での計算

実行結果

(my-length (list 11 12)) から 2 が得られる過程の概略

(my-length (list 11 12))

= ...

= (+ 1 (my-length (list 12)))

= ...

= (+ 1 (+ 1 (my-length empty)))

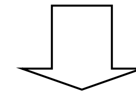
= ...

= (+ 1 (+ 1 0))

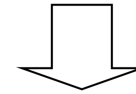
= (+ 1 1)

= 2

(list 11 12) の長さ



(list 12) の長さに 1 を足す



empty の長さに 2 を足す

よくある勘違い

```
(define (my-length a-list)
   (cond
      [(= a-list empty) 0]
      [else (+ 1
               (my-length (rest a-list)))]))
```

終了条件の判定：
・ 正しくは「(empty? a-list)」
・ x がリストのとき、(= x empty) は実行エラー
・ 「=」は数値の比較には使えるが，リスト同士の比較には使えない

実行エラーの例

Untitled - DrScheme

File Edit Windows Show Language Scheme Help

Untitled | (define ...) | Save | Check Syntax | Step | Execute | Break

```
(define (list-length a-list)
  (cond
    [(= a-list empty) 0]
    [else (+ 1
             (list-length (rest a-list)))]))
```

```
> (list-length (list 11 12))
=: expects type <number> as 2nd argument,
given: empty; other arguments were: (list 11
12)
>
```

7:3 Unlocked not running

リストの長さ

1. リストが空ならば：　→　終了条件

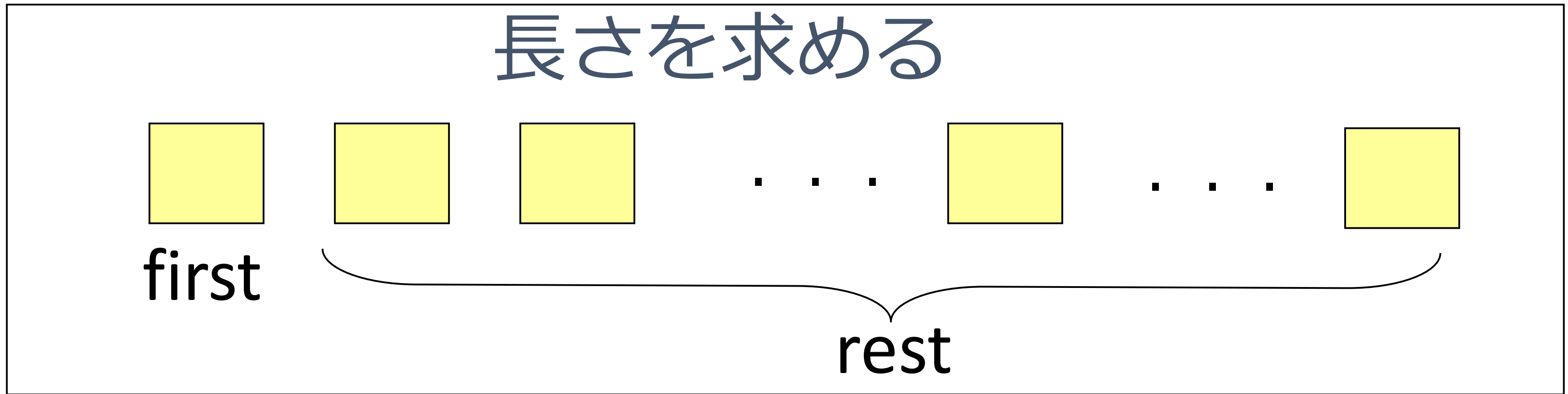
    0　→　自明な解

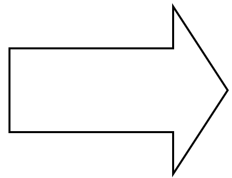
2. そうで無ければ：

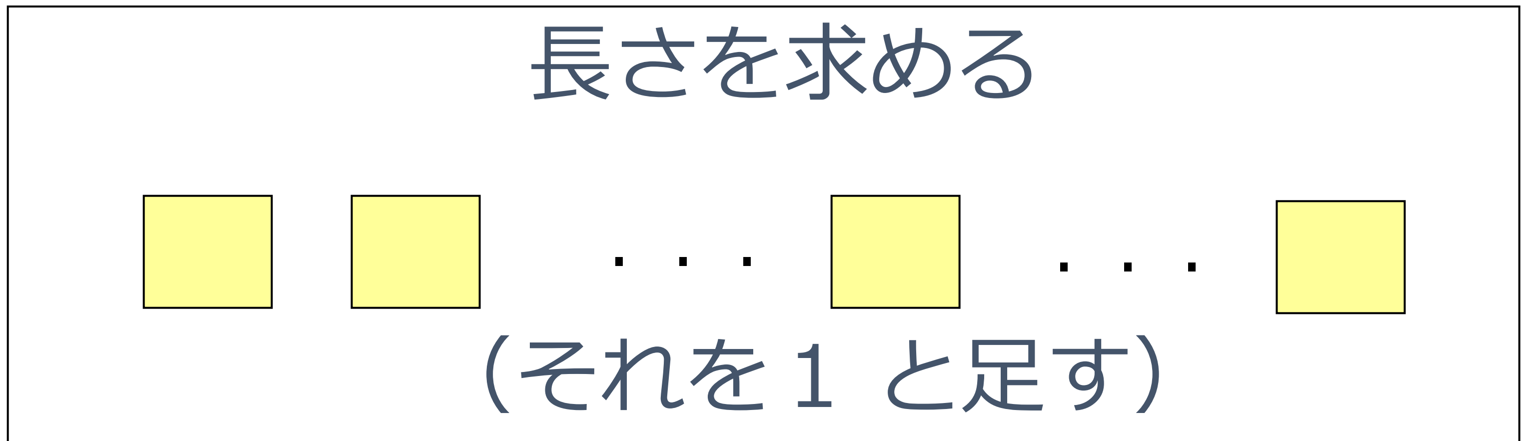
    - リストの rest を求める（これもリスト）．「その長さに１を足したもの」が，求める総和である

リストの長さ

リストが空で無いとき





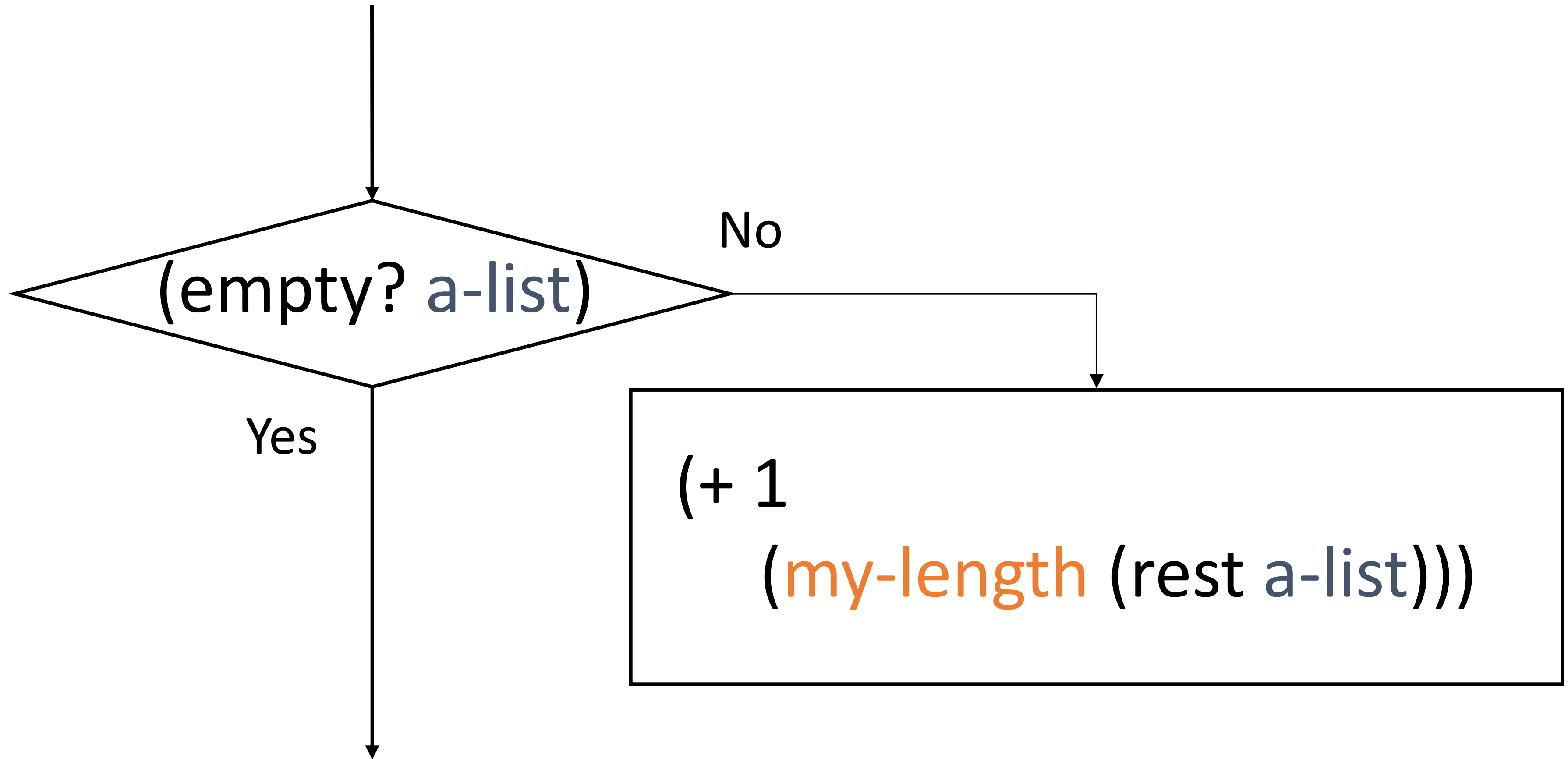


リストの長さ

```
(define (my-length a-list)
  (cond
    [(empty? a-list) 0]
    [else (+ 1
            (my-length (rest a-list)))]))
```

終了条件: (empty? a-list)

自明な解: 0



0 が自明な解である

リストの長さ my-length

- my-length の内部に my-length が登場

```
(define (my-length a-list)
  (cond
    [(empty? a-list) 0]
    [else (+ 1
            (my-length (rest a-list)))]))
```

- my-length の実行が繰り返される

  例： (my-length (list 11 12))

  = = (+ 1 (my-length (list 12)))